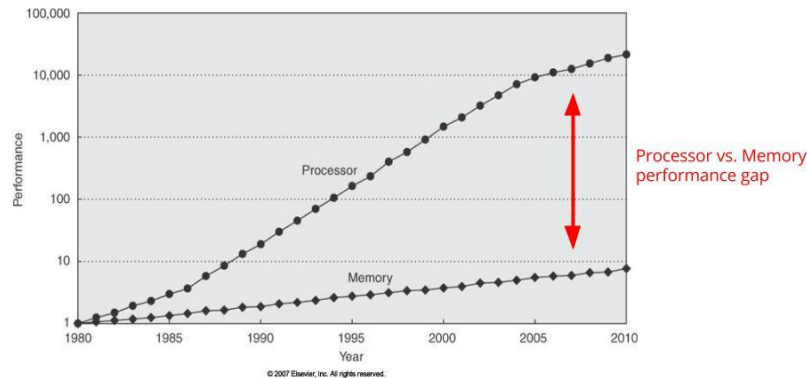


CPU Cache: Overview and Optimisation

Michael Rooney

What's wrong with using RAM?

- Before explaining what a cache does, I will first give a quick overview of why normal RAM accesses are unacceptably slow
- RAM performance has not improved at the same rate as CPU performance
- It can take over 40-80ns to just request data from RAM
- Most modern consumer and professional CPUs have a clock speed between 2-4GHz, which means one clock cycle takes 0.25-0.5ns
- If the CPU must wait until the requested data is retrieved from memory then hundreds to thousands of cycles are spent doing nothing



Memory speed lags behind CPU speed

Source: dave.cheney.net/2014/06/07/five-things-that-make-go-fast

What is a cache?

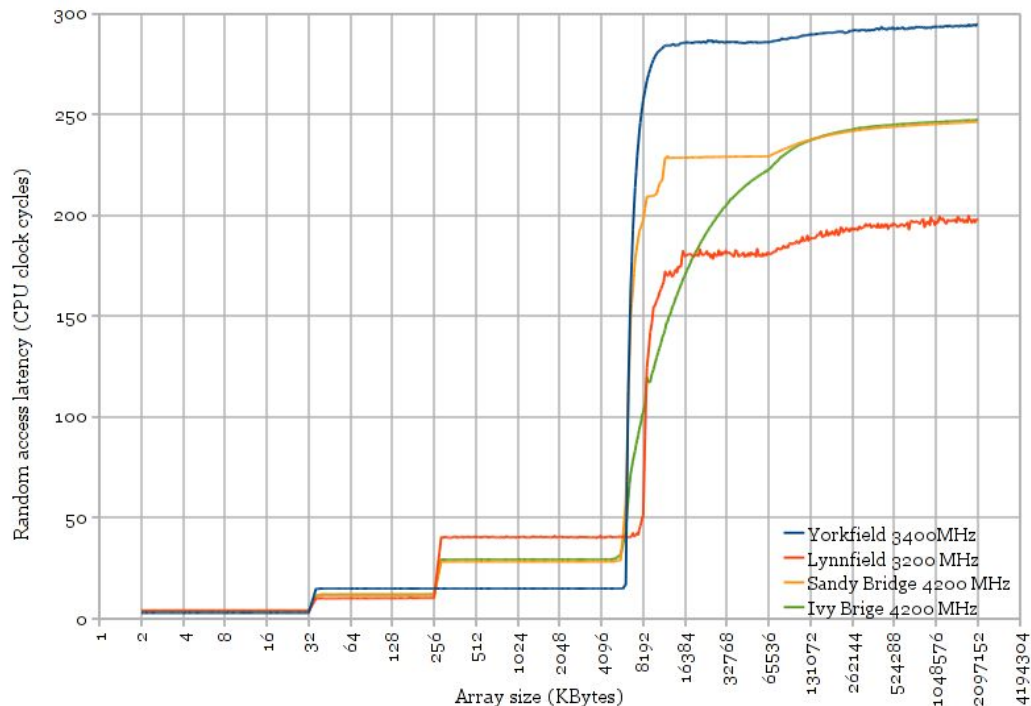
- A cache is a very fast memory that exists on the CPU chip
- Typically divided into three “levels” in modern CPUs.
- Each successive level has a higher capacity, but also a higher access time than the previous level
- Usually the highest level is shared between all cores.
- A cache’s associativity determines how many places an address can map to, but increasing the associativity increases the access time as more locations have to be searched for the address.
- Typically L1 cache has a lower associativity (2-8x), and higher cache levels have a higher associativity (8-16x).
- This is to keep access times low for L1 cache, while having a higher hit rate for higher cache levels
- The cache will attempt to exploit spatial and temporal locality.
- Temporal locality means that if an address is used it is likely to be used again soon, so the data at that address will be kept in the cache.
- Spatial locality means that if an address is used it is likely that nearby addresses will also be used, so the cache stores the values in these nearby addresses

The memory hierarchy

<u>Name</u>	<u>Size</u>	<u>Access latency</u>
L1 cache	16-64KB	1ns
L2 cache	256-1024KB	3-10ns
L3 cache	2MB-60MB	10-20ns
RAM	1GB-3TB	40-80ns
Solid state drive	40GB-4TB	0.1-1ms
Hard drive	500GB-10TB	5-10ms

These figures can vary heavily depending on the exact hardware. However these figures demonstrate the order of magnitude for each field

Access times vs size



Source: <http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/>

How can I profile my cache usage? (1)

One option is valgrind's cachegrind tool

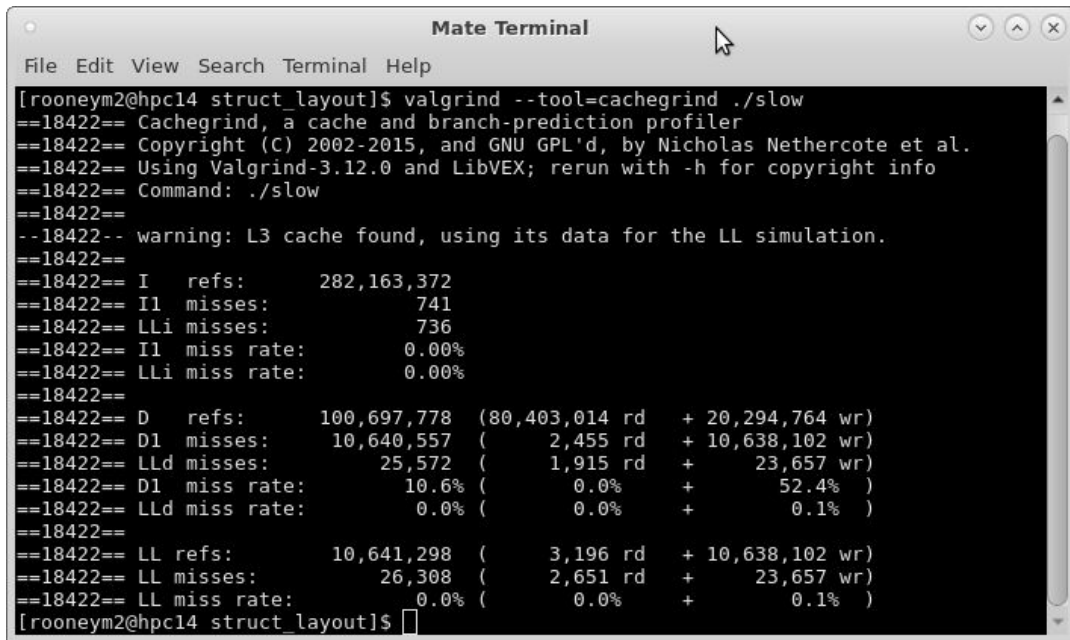
```
valgrind --tool=cachegrind ./program
```

For L1 cache shows results for instruction (I) and data (D).

Shows overall results for LL cache, which represents the last level cache.

By default the size, line size and associativity of each cache level is set to the match that of the host's CPU.

Size, line size, associativity can be changed via passed arguments.



```
Mate Terminal
File Edit View Search Terminal Help

[rooneym2@hpc14 struct_layout]$ valgrind --tool=cachegrind ./slow
==18422== Cachegrind, a cache and branch-prediction profiler
==18422== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==18422== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==18422== Command: ./slow
==18422==
--18422-- warning: L3 cache found, using its data for the LL simulation.
==18422==
==18422== I   refs:      282,163,372
==18422== I1 misses:      741
==18422== LLi misses:      736
==18422== I1 miss rate:    0.00%
==18422== LLi miss rate:  0.00%
==18422==
==18422== D   refs:      100,697,778 (80,403,014 rd + 20,294,764 wr)
==18422== D1 misses:      10,640,557 ( 2,455 rd + 10,638,102 wr)
==18422== LLd misses:      25,572 ( 1,915 rd + 23,657 wr)
==18422== D1 miss rate:    10.6% ( 0.0% + 52.4% )
==18422== LLd miss rate:  0.0% ( 0.0% + 0.1% )
==18422==
==18422== LL refs:      10,641,298 ( 3,196 rd + 10,638,102 wr)
==18422== LL misses:      26,308 ( 2,651 rd + 23,657 wr)
==18422== LL miss rate:    0.0% ( 0.0% + 0.1% )
[rooneym2@hpc14 struct_layout]$
```

How can I profile my cache usage? (2)

Cachegrind also outputs a file that can be used to get cache usage statistics for each line of your source code.

However Cachegrind does have some limitations.

It simulates a CPU that runs your code, so the results will not exactly match what can be expected on your target platform.

It doesn't account for system calls, or for other programs that are also running on the processor.

You can't completely rely on the results, but it is definitely worthwhile to consider them.

It also seems to have trouble with large memory allocations.

As a side note: because it simulates a CPU it can be quite slow. Because of this and its issues with large memory allocations, I recommend that you create a small program that demonstrates the core of your algorithm and data structures.

How can I optimise my cache usage? (1)

How can I optimise my cache usage? (1)

Don't

How can I optimise my cache usage? (1)

Don't - unless there is a good reason to

Profile your code first - find out if the cache usage is a problem

Optimising for cache usage usually involves trade offs

- It takes up time that could be spent improving other parts of the code
- Code becomes more complex and harder to work with
- Sometimes additional (possibly expensive!) steps are needed to lay out the data in a cache friendly way

Ensure that the cache optimisations would have a benefit, and that the tradeoffs are worth it.

How can I optimise my cache usage? (2)

I will give examples of some ways of optimising cache usage. These examples are not intended to be copied directly - but instead should give you ideas to start with.

Some of these examples are slightly contrived - can't fit a 10'000 lines of code real world example in a slideshow.

In general you need to look at where your data is being stored, and how it is being accessed. From there you can decide on the best approach. The exact approach to improve cache usage is highly dependent on the particular program's algorithms and data formats. Some possible approaches are:

- Rearrange data structures to keep frequently accessed data close together.
- Replace large data types with smaller data types when possible.
- If possible use arrays instead of linked lists.
- Use a single large malloc call instead of many small malloc calls
- Use pointers instead of arrays in structs.

Example 1: matrix multiplication (1)

Scenario:

Your program multiplies two large matrices of any size. The matrices are stored in row-major format (rows are adjacent in memory), which is standard in C.

Problem:

During multiplication we make strided accesses to the B matrix, e.g. we access row 0 column 0, then row 1 column 0 and so on.

However as the matrix is in row major format this means that we are “jumping over” the rest of the row when we accessing the next row.

The cache is loading the rest of the row in an attempt to exploit spatial locality, but we are not using it (possibly it will be used in the future though).

Solution:

Create a transposed copy of the B matrix and adjust the access to $B[j][k]$ instead of $B[k][j]$. Now B's column data is stored sequentially in memory and we make use of spatial locality.

Downsides:

The cost of transposing the matrix may outweigh the cache performance benefits (especially for oddly shaped matrices).

Need to check whether or not it is worthwhile to transpose the matrix.

Results:

When multiplying two 1000x1000 matrices I recorded the following times:

CPU	Intel i5 2500k (single threaded)	AMD 64 core server (multi threaded)
Original time (seconds)	10.31	0.461
Cache optimised time (seconds)	3.53 (2.9x faster)	0.087 (5.3x faster)

Example 1: matrix multiplication (2)

Matrix A		
a	b	c
d	e	f
g	h	i



Matrix A in memory								
a	b	c	d	e	f	g	h	i

With a matrix with very large rows the cache cannot load the next row.

As we are accessing only one element of each row at a time in matrix B, the cache's attempts to exploit spatial locality are wasted. Note: for small matrices the cache will likely load the next desired column element anyway since the data is overall small. Try to visualise this problem but with a very large matrix

Matrix B		
j	k	l
m	n	o
p	q	r



Matrix B in memory								
j	k	l	m	n	o	p	q	r

Example 2: single large malloc call for linked lists (1)

Scenario:

Your program makes many malloc calls to create nodes in a frequently used linked list. Nodes are rarely deleted.

Problem:

The C standard does not guarantee that consecutive malloc calls return adjacent addresses.

Two consecutive malloc calls could return vastly different addresses.

Even if your C library's malloc implementation does return adjacent addresses it is likely that you'll be making other calls to malloc in between allocating nodes for the linked list.

Both of these issues will cause your linked list's nodes to become disjoint in memory, and will lead to poor cache usage.

Solution:

Make a single large malloc call and manually hand out memory address when a new node is needed.

This large block of memory is the "pool".

Track how much of the pool has been used, and make another pool if more space is needed.

This acts somewhat like an array, but because it can be broken into multiple pools you don't need to know the size in advance.

In addition it does not need to be in a large contiguous memory space like an array does. This can become an issue when working with very large data sets.

Example 2: single large malloc call for linked lists (2)

Downsides:

Complicates freeing memory if a node is no longer needed.
You need to ensure the old data has been deleted when re-using addresses from the pool.
Requires extra work to manage the pool.

Testing:

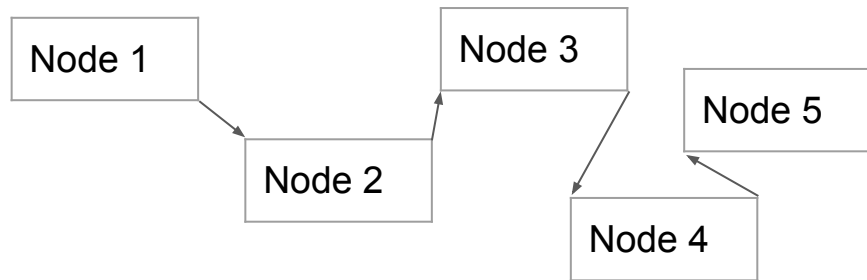
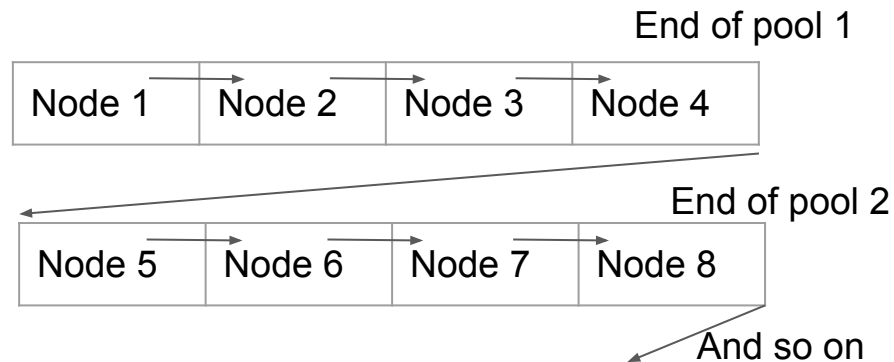
Linked lists with 1000 nodes were traversed 10'000 times.
My particular version of the C library did return adjacent addresses.
To simulate nodes being disjoint in memory I inserted padding between the nodes.

Results:

Normal linked list: 12.5% D1 miss rate, approximately 0.11 second run time
Pooled linked list: < **0.1%** D1 miss rate, approximately 0.071 second run time

Around a 35% reduction in time taken.

Example with pools of size 4



Example 3: struct with a rarely used array/vector (1)

Scenario:

Your program uses an array of structs. These structs contain a vector and other data. The array is frequently iterated over, but the vector field is rarely used.

Problem:

The structs are adjacent in memory, but because of the rarely used vector the cache will not actually load the data that is used next.

Solution:

Replace the vector field with a pointer, so that it will only take up 8 bytes.

Downsides:

Requires an extra malloc call when the struct is created.

Requires an extra free call when the struct is deleted.

Takes more time when the array field is used.

```
struct slow {  
    int a;  
    int b;  
    char c[100]  
}
```

```
struct fast {  
    int a;  
    int b;  
    char *c;  
}
```

array[0]			array[1]		
a	b	c[0] ... c[99]	a	b	c[0] ...c[99]

Example 3: struct with a rarely used array/vector (2)

Testing:

Two arrays with 10'000 elements were created, one using the struct with a vector and one using the struct with a pointer.

The size of the vector was set to 100 bytes.

The arrays were iterated over 1'000 times.

The a and b fields of each struct were modified on each iteration.

Results:

Using struct with vector: 10.6% D1 miss rate, approximately 0.131 second runtime

Using struct with pointer: 2.5% D1 miss rate, approximately 0.071 second runtime

Almost half as much time taken.

Example 4: struct with many booleans (1)

Quick note for new programmers: a boolean is a data type that can only be true or false, and in C bitwise operators such as OR, AND, and others can be used to interact with the bits of a variable.

Scenario:

Your program has a struct that has many boolean fields, as well as some other fields. A list of these structs is frequently traversed, and the booleans in each struct are set and/or checked.

Problem:

Booleans require only a single bit to store their value. However they will use at least 1 byte of memory, but may use up to 4 bytes of memory depending on the particular system. This is 8x-32x as much space as they actually need. This leads to poor cache performance as the cache is saving useless data.

Solution:

An int variable has 32 bits. Using masks and bitwise operators we can set and read individual bits of an int.

To set bit N of an int (counting from 0), use a mask = 2^N and use the OR operator.

To read bit N of an int (counting from 0), use mask = 2^N and use the AND operator, then check if the result is > 0 .

This allows us to store 32 booleans in a single integer.

To set bit 5: `bitfield = bitfield | 32;`

To read bit 5: `unsigned int result = bitfield & 32;`

Example 4: struct with many booleans (2)

Downsides:

Complicates the code as masks will need to be defined somewhere.

Masks need careful naming so the programmer knows what bit each individual boolean corresponds to.

More instructions required to access/modify the individual bits of the int.

Testing:

An array of 1'000'000 structs each containing 32 booleans and a 3 element int array was traversed 100 times, with 2 booleans accessed each time.

The test was repeated for a new struct which contains one int that acts as a bit field and a 3 element int array.

Results:

Using 32 booleans: 8.6% D1 miss rate, approximately 0.335 second runtime.

Using single int: 1.8% D1 miss rate, approximately 0.264 second runtime.

A little over 20% reduction in time taken.

```
struct test_bool {  
    bool values[32];  
    int vec[3];  
};
```

```
struct test_int {  
    int values;  
    int vec[3];  
};
```

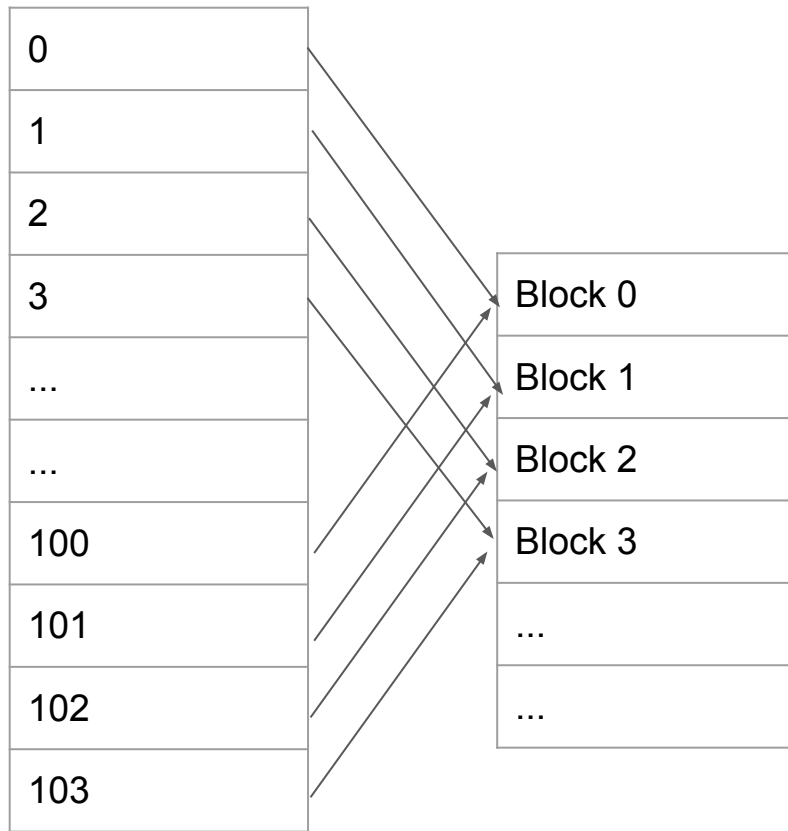
Other issues to watch out for

It's unlikely but possible that your algorithm may be accessing data multiple times, but at a rate that exceeds the lifetime of the data in the cache.

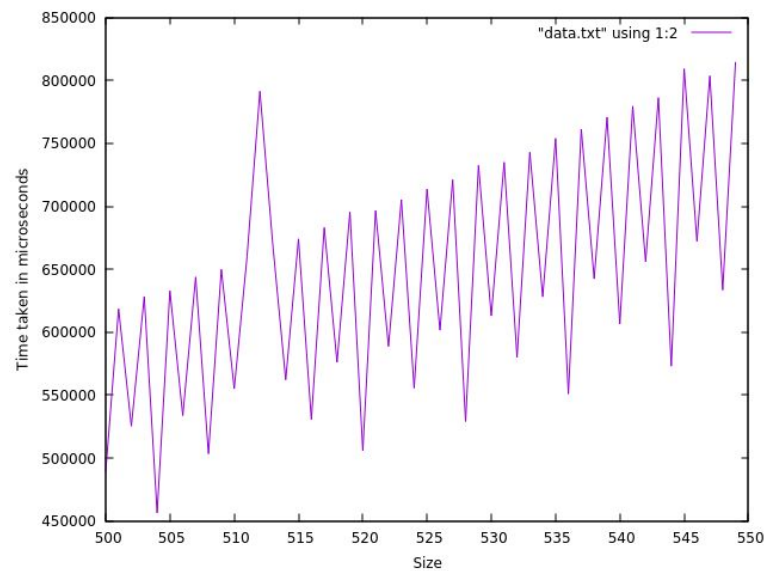
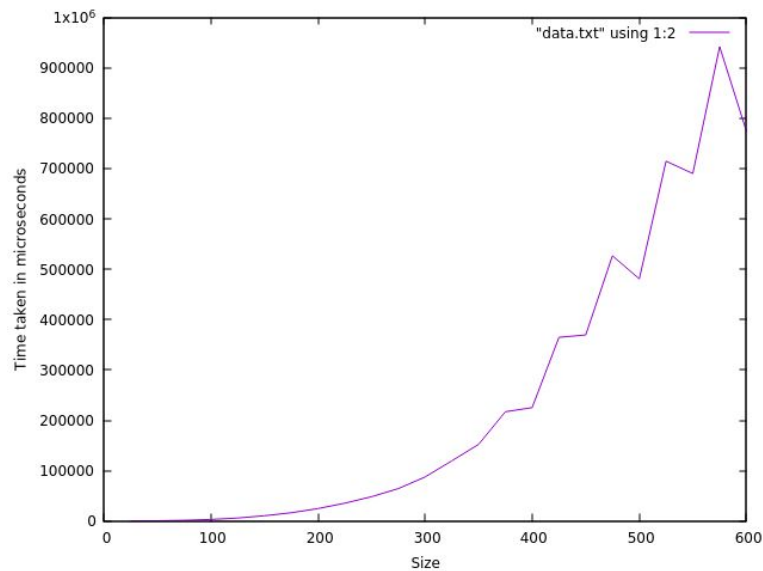
To fix this you could add in dummy reads, to ensure that the data is not the least recently used.

Another issue is cache trashing. This happens when you access data in such a way that causes many pieces of data to compete for the same location in the cache.

To fix this you should change your algorithm (aka change your data access pattern), or change the layout of data in memory.



The effects of cache trashing



Optimisations that may hurt cache performance

There are some optimisations that can hurt cache performance, while possibly increasing overall performance. I will describe two of them.

Function inlining is an optimisation that replaces function calls with a copy of the function, hopefully saving time by avoiding the need to call the function and set up the parameters. Usually it is done by the compiler.

Loop unrolling reduces the loop control overhead by reducing the total number of iterations by doing more work per iteration. Loop can easily be done manually, but the compiler can also do it.

These can cause worse instruction cache performance, as they make the executable larger. You should check if the overall time saved makes up for any impact on cache performance.

```
int square_func(int x){
    return x*x;
}

int foo(int x){
    int a = square_func(x);
    // blah blah blah
}

int foo(int x){
    int a = x*x; // inlined
    // blah blah blah
}

int i;
for(i = 0; i < 100; i = i + 5){
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
    a[i+4] = b[i+4] + c[i+4];
}
```

Slides and code examples can be found at:

<https://github.com/MikeyRooney/MA5691>